Data Allocation Strategies for Dense Linear Algebra Kernels on Heterogeneous Two-dimensional Grids

Vincent Boudet, Antoine Petitet, Fabrice Rastello, Yves Robert

Data Allocation Strategies for Dense Linear Algebra Kernels on Heterogeneous Two-dimensional Grids.

Vincent Boudet
Antoine Petitet
Fabrice Rastello
Yves Robert

June 1999

Research Report N^o 99-31

Data Allocation Strategies for Dense Linear Algebra Kernels on Heterogeneous Two-dimensional Grids.

Vincent Boudet
Antoine Petitet
Fabrice Rastello
Yves Robert

June 1999

Abstract

We study the implementation of dense linear algebra computations, such as matrix multiplication and linear system solvers, on two-dimensional (2D) grids of heterogeneous processors. For these operations, 2D-grids are the key to scalability and efficiency. The uniform block-cyclic data distribution scheme commonly used for homogeneous collections of processors limits the performance of these operations on heterogeneous grids to the speed of the slowest processor. We present and study more sophisticated data allocation strategies that balance the load on heterogeneous 2D-grids with respect to the performance of the processors. The practical usefulness of these strategies is fully demonstrated by experimental data for a heterogeneous network of workstations.

Keywords: heterogeneous network, heterogeneous grid, different-speed processors, load-balancing, data distribution, data allocation, numerical libraries.

Résumé

Dans ce rapport, nous étudions l'implémentation de programmes d'algèbre linéaire, tels que la multiplication de matrices ou la résolution de systèmes linéaires, sur une grille hétérogène bidimensionnelle de processeurs. Pour ces problèmes, seule une grille 2D assure la scalabilité des algorithmes utilisés. La distribution classique "bloc-cyclique" utilisée communément dans le cas d'une grille homogène de processeurs, réduit la performance sur une grille hétérogène à la vitesse du processeur le plus lent. L'intéret pratique de notre étude est grandement justifié par des experiences effectuées sur un réseau local de machines hétérogènes.

Mots-clés: plateforme hétérogène, grille hétérogène, processeurs de vitesses différentes, distribution des données, équilibrage de charges, librairies de calcul.

1 Introduction

Heterogeneous networks of workstations (HNOWs) are ubiquitous in university departments and companies. They represent the typical poor man's parallel computer: running a large PVM or MPI experiment (possibly all night long) is a cheap alternative to buying supercomputer hours. The idea is to make use of *all* available resources, namely slower machines *as well as* more recent ones. In addition, parallel machines used in a multi-user environment exhibit the very characteristics of a HNOW: different loads imply different processor speeds when running a parallel application, even though all processors are identical. Note that multi-user parallel machines are interesting in this context because they may exhibit a better communication-to-computation ratio than Ethernet-based networks.

The major limitation to programming heterogeneous platforms arises from the additional difficulty of balancing the load when using processors running at different speed. In this paper, we explore several possibilities to implement linear algebra kernels on HNOWs. We have been exploring data allocation strategies for HNOWs arranged as a uni-dimensional (linear) array in previous papers [7, 8]. Arranging the processors along a two-dimensional grid turns out to be surprisingly difficult. There are two complicated problems to solve: (i) how to arrange the heterogeneous processors along a 2D-grid; (ii) how to distribute matrix blocks to the processors once the grid is built. The major contribution of this paper is to provide an efficient solution to both problems, thereby providing the required framework to build an extension of the ScaLAPACK library [6] capable of running on top of HNOWs or non-dedicated parallel machines.

The rest of the paper is organized as follows. In Section 2 we discuss the framework for implementing our heterogeneous kernels, and we briefly review the existing literature. In Section 3 we summarize existing algorithms for matrix multiplication and dense linear solvers on 2D (homogeneous) grids. In Section 4 we propose data allocation strategies for implementing the previous kernels on 2D heterogeneous grids. In Section 5 we present experimental results that demonstrate the practical usefulness of these strategies on a HNOW. We give some final remarks and conclusions in Section 6.

2 Framework

2.1 Static Versus Dynamic Strategies

Because we have a library designer's approach, we target static strategies to allocate data and computations to the processors. In fact, distributing the computations (together with the associated data) can be performed either dynamically or statically, or a mixture of both. On one hand, we may think that dynamic strategies are likely to perform better, because the machine loads will be self-regulated, hence self-balanced, if processors pick up new tasks just as they terminate their current computation. However, data dependences, communication costs and control overhead may well lead to slow the whole process down to the pace of the slowest processors. On the other hand, static strategies will suppress (or at least minimize) data redistributions and control overhead during execution. Furthermore, in the context of a numerical library, static allocations seem to be necessary for a simple and efficient data allocation.

However, to be successful, static strategies must obey a more refined model than standard block-cyclic distributions: such distributions are well-suited to processors of equal speed but lead to a great load imbalance between processors of different speed.

2.2 Machine Model

Our machine model is either a heterogeneous network of workstations of different speeds, or a parallel computer with multiple users. In this latter case, we assume different loads on the different processors, thereby considering the parallel computer as a heterogeneous machine (and calling it a HNOW as well). In both cases, we have to model the communication links. For HNOWS interconnected with a standard Ethernet network, all communications are inherently sequential, while for Myrinet or switched networks, independent communications can take place in parallel. In all cases, we consider that the communications performed by one processor are sequential.

In any case, we will configure the HNOW as a (virtual) 2D grid for scalability reasons [9]. We come back to this point in Section 3 when describing the well-known blocked matrix multiplication and LU or QR decomposition algorithms.

2.3 Load-balancing for Uni-dimensional Grids

We briefly survey our previous work [7, 8] in this section. We consider a linear array of heterogeneous processors.

2.3.1 Matrix-matrix Multiplication

The matrix-matrix multiplication algorithm basically reduces to a succession of steps composed of independent computations on each processor [21].

To efficiently parallelize the matrix-matrix multiplication algorithm we therefore have to solve the following problem: given M independent chunks of computations, each requiring the same amount of work, how can we assign these chunks to p processors $P_1, P_2, \cdots, P_p$ of respective execution times $t_1, t_2, \cdots, t_p$ so that the load is best balanced ? Intuitively, the load of P_i should be inversely proportional to t_i: P_i receives $c_i = \frac{\frac{1}{t_i}}{\sum_{j=1}^{p} \frac{1}{t_j}} \times M$ chunks. This strategy leads to a perfect load balance when M is a multiple of $C = \operatorname{lcm}(t_1, t_2, \cdots, t_p) \sum_{j=1}^{p} \frac{1}{t_j}$, a quantity that may be very large. For the general case, the following algorithm provides the best solution [8]:

Algorithm 2.3.1: Optimal distribution for M independent chunks, over p processors of cycle-times $t_1, \ldots, t_p$

```
# Initialization: Approximate the $c_i$ so that $c_i \times t_i \approx Constant$, and $c_1 + c_2 + \ldots + c_p \leq M$.
$forall\, i \in \{1, \ldots, p\}$, $c_i = \left\lfloor \frac{\frac{1}{t_i}}{\sum_{i=1}^{p} \frac{1}{t_i}} \times M \right\rfloor$.
# Iteratively increment some $c_i$ until $c_1 + c_2 + \ldots + c_p = M$
for $m = c_1 + c_2 + \ldots + c_p$ to M
    find $k \in \{1, \ldots, p\}$ such that $t_k \times (c_k + 1) = \min\{t_i \times (c_i + 1))\}$
    $c_k = c_k + 1$
```

Consider a toy example with 3 processors of relative cycle-time $t_1 = 3$, $t_2 = 5$ and $t_3 = 8$. To allocate a total of $M = 10$ chunks, Algorithm 2.3.1 assigns 5 chunks to P_1, 3 chunks to P_2 and 2 chunks to P_3.

2.3.2 The LU and QR Decompositions

The (blocked) LU and QR decomposition algorithms work as follows on a linear array: at each step, the pivot processor processes the pivot panel (a block of r columns) and broadcasts it to all the processors, which update their remaining columns. For next step, the next r columns become the pivot panel, and the computation progresses. The preferred distribution for a homogeneous NOW is a $CYCLIC(r)$ distribution of columns, where optimal values of r depend on the communication-to-computation ratio of the target computer.

Consider the first step. After the factorization of the first block, all updates are independent chunks: here a chunk consists of the update of a single block of r columns. If the matrix size is $n = M \times r$, there are $M - 1$ chunks. We can use Algorithm 2.3.1 to distribute these independent chunks. The size of the trailing submatrix shrinks as the computation goes on. At the second step, the number of blocks to update is only $M - 2$. If we want to distribute these chunks independently of the first step, redistribution of data will have to take place between the two steps, and this will incur a lot of communications. Rather, we search for a static allocation of columns blocks to processors that will remain the same throughout the computations, as the elimination progresses. We aim at balancing the updates of all steps with the same allocation: in other words, we need a distribution that is kind of repetitive (because the matrix shrinks) but not fully cyclic (because processors have different speeds).

Looking closer at the successive updates, we see that only column blocks of index $i+1$ to M are updated at step i. Hence our objective is to find a distribution such that for each $i \in \{2, \ldots, M\}$, the amount of blocks in $\{i, \ldots, M\}$ owned by a given processor is approximately proportional to its speed. To derive such a distribution, we use a dynamic programming algorithm which is best explained using a toy example:

Table 1: Running the dynamic programming algorithm with 3 processors: $t_1 = 3$, $t_2 = 5$, and $t_3 = 8$.

Number of chunks	c_1	c_2	c_3	Average cost	Selected processor
0	0	0	0		1
1	1	0	0	3	2
2	1	1	0	2.5	1
3	2	1	0	2	3
4	2	1	1	2	1
5	3	1	1	1.8	2
6	3	2	1	1.67	1
7	4	2	1	1.71	1
8	5	2	1	1.87	2
9	5	3	1	1.67	3
10	5	3	2	1.6	

In Table 1, we report the allocations found by the algorithm up to $B = 10$. The entry "Selected processor" denotes the rank of the processor chosen to build the next allocation. At each step, "Selected processor" is computed so that the cost of the allocation is minimized. The cost of the allocation is computed as follows: the execution time, for an allocation $\mathcal{C} = (c_1, c_2, \ldots, c_p)$ is $\max_{1 \leq i \leq p} c_i t_i$ (the maximum is taken over all processor execution times), so that the average cost to execute one chunk is

$$cost(\mathcal{C}) = \frac{\max_{1 \leq i \leq p} c_i t_i}{\sum_{i=1}^{p} c_i}$$

For instance at step 4, i.e. to allocate a fourth chunk, we start from the solution for three chunks, i.e. $(c_1, c_2, c_3) = (2, 1, 0)$. Which processor P_i should receive the fourth chunk, i.e. which c_i should be incremented? There are three possibilities $(c_1 + 1, c_2, c_3) = (3, 1, 0)$, $(c_1, c_2 + 1, c_3) = (2, 2, 0)$ and $(c_1, c_2, c_3 + 1) = (2, 1, 1)$ of respective costs $\frac{9}{4}$ (P_1 is the slowest), $\frac{10}{4}$ (P_2 is the slowest), and $\frac{8}{4}$ (P_3 is the slowest). Hence we select $i = 3$ and we retain the solution $(c_1, c_2, c_3) = (2, 1, 1)$.

Back to the LU and QR decompositions, we allocate slices of B consecutive blocks to processors, as illustrated in Figure 1. B is a parameter to be provided by the user. For a matrix of size $n = m \times r$, we can simply let $B = m$, i.e. define a single slice. But we can prefer to have a fixed B, say $B = 10$, to have a generic allocation that only depends upon the processor speeds, not upon the problem size.

Within each slice, we use the dynamic programming algorithm for $s = 0$ to $s = B$ in a "reverse" order. Consider again the toy example with 3 processors of relative cycle-time $t_1 = 3$, $t_2 = 5$ and $t_3 = 8$. The dynamic programming algorithm allocates chunks to processors as shown in Table 2. The allocation of chunks to processors is obtained by reading the second line of Table 2 from right to left: $(3, 2, 1, 1, 2, 1, 3, 1, 2, 1)$ (see Figure 2 for the detailed allocation within a slice). As illustrated in Figure 1, at a given step there are several slices of at most B chunks, and the number of chunks in the first slice decreases as the computation progresses (the leftmost chunk in a slice is computed first and then there only remains $B - 1$ chunks in the slice, and so on). In the example, the reversed allocation best balances the update in the first slice at each step: at the first step when there are the initial 10 chunks (1 factor and 9 updates), but also at the second step when only 8 updates remain, and so on. The updating of the other slices remains well-balanced by construction, since their size does not change, and we keep the best allocation for $B = 10$. See Figure 2 for the detailed allocation within a slice, together with the cost of the updates.

Table 2: Static allocation for $B = 10$ chunks.

Chunk number	1	2	3	4	5	6	7	8	9	10
Processor number	1	2	1	3	1	2	1	1	2	3

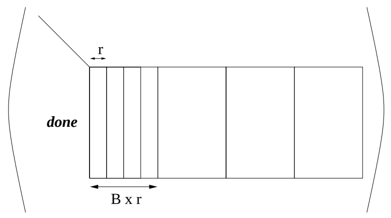

Figure 1: Allocating slices of B chunks.

2.4 Related Work

There is a great deal of papers in the literature dealing with dynamic schedulers to distribute the computations (together with the associated data) onto heterogeneous platforms. Most schedulers use naive mapping strategies such as master-slave techniques or paradigms based upon the idea *"use the past to predict the future"*, i.e. use the currently observed speed of computation of each machine to decide for the next distribution of work: see the survey paper of Berman [4] and the more specialized references [2, 12] for further details. Several scheduling and mapping heuristics have been proposed to map task graphs onto HNOWs [24, 25, 22, 19]. Scheduling tools such as Prophet [26] or AppLeS [4] are available (see also the survey paper [23]).

The static mapping of numerical kernels has however received much less attention. To the best of our knowledge, there is a single paper by Kalinov and Lastovetky [20] which has similar objectives as ours. They are interested in LU decomposition on heterogeneous 1D and 2D grids. They propose a "heterogeneous block cyclic distribution" to map matrix blocks onto the different-speed processors. They use the *mPC* programming tool [3] to program the heterogeneous 1D or 2D grids, which they consider as fixed (they do not discuss how to configure the grid). In fact, their "heterogeneous block cyclic distribution" does not lead to a "true" 2D-grid, because each processor has more than 4 direct neighbors to communicate with. We come back on this distribution in Section 3.1.2.

3 Linear Algebra Kernels on 2D Grids

In this section we briefly recall the algorithms implemented in the ScaLAPACK library [6] on 2D homogeneous grids. Then we discuss how to modify the two-dimensional block-cyclic distribution which is used in ScaLAPACK to cope with 2D heterogeneous grids.

3.1 Matrix-matrix Multiplication

3.1.1 Homogeneous Grids

For the sake of simplicity we restrict to the multiplication $C = AB$ of two square $n \times n$ matrices A and B. In that case, ScaLAPACK uses the outer product algorithm described in [1, 16, 21]. Consider a 2D processor grid of size $p \times q$.

Assume first that $n = p = q$. In that case, the three matrices share the same layout over the 2D grid: processor $P_{i,j}$ stores $a_{i,j}$, $b_{i,j}$ and $c_{i,j}$. Then at each step k,

- each processor $P_{i,k}$ (for all $i \in \{1, .., p\}$) horizontally broadcasts $a_{i,k}$ to processors $P_{i,*}$.
- each processor $P_{k,j}$ (for all $j \in \{1, .., q\}$) vertically broadcasts $b_{k,j}$ to processors $P_{*,j}$.

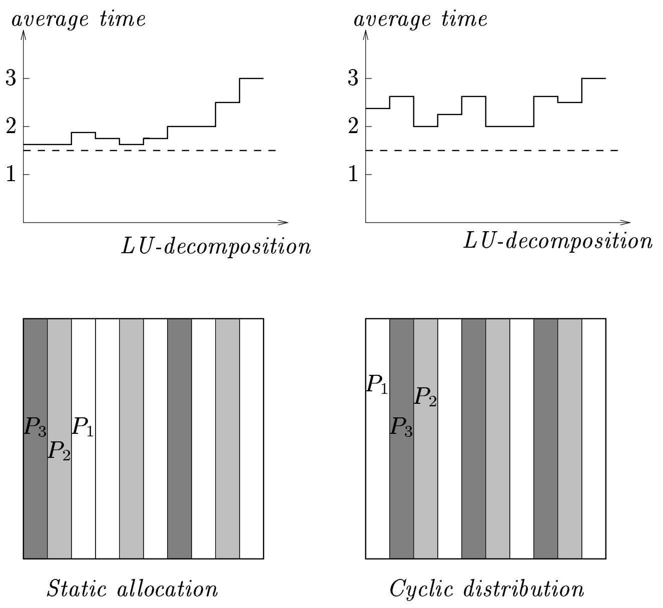

Figure 2: Comparison of two different distributions for the LU-decomposition algorithm on a heterogeneous platform made of 3 processors of relative cycle-time 3, 5 and 8. The first distribution is the one given by our algorithm, the second one is the cyclic distribution. The total number of chunks is $B = 10$.

so that each processor $P_{i,j}$ can independently compute $c_{i,j} += a_{i,k} \times b_{k,j}$.

This algorithm is used in the current version of the ScaLAPACK library because it is scalable, efficient and it does not need any initial permutation (unlike Cannon's algoritm [21]). Moreover, on a homogeneous grid, broadcasts are performed as independent ring broadcasts (along the rows and the columns), hence they can be pipelined.

Of course, ScaLAPACK uses a blocked version of this algorithm to squeeze the most out state-of-the-art processors with pipelined arithmetic units and multilevel memory hierarchy [15, 9]. Each matrix coefficient in the description above is replaced by a $r \times r$ square block, where optimal values of r depend on the communication-to-computation ratio of the target computer.

Finally, a level of virtualization is added: usually, the number of blocks $\lceil \frac{n}{r} \times \rceil \lceil \frac{n}{r} \rceil$ is much greater than the number of processors $p \times q$. Thus blocks are scattered in a cyclic fashion along both grid dimensions, so that each processor is responsible for updating several blocks at each step of the algorithm. An example is given in Figure 3 with $p = q = 4$ and $\lceil \frac{n}{r} \rceil = 10$.

1	2	3	4	1	2	3	4	1	2
5	6	7	8	5	6	7	8	5	6
9	10	11	12	9	10	11	12	9	10
13	14	15	16	13	14	15	16	13	14
1	2	3	4	1	2	3	4	1	2
5	6	7	8	5	6	7	8	5	6
9	10	11	12	9	10	11	12	9	10
13	14	15	16	13	14	15	16	13	14
1	2	3	4	1	2	3	4	1	2
5	6	7	8	5	6	7	8	5	6

Figure 3: Processors are numbered from 1 to 16. This figure represents the distribution of 10×10 matrix blocks onto 4×4 processors.

3.1.2 Heterogeneous Grids

Suppose now we have a $p \times q$ grid of heterogeneous processors. Instead of distributing the $r \times r$ matrix blocks cyclically along each grid dimension, we distribute *block panels* cyclically along each grid dimension. A block panel is a rectangle of consecutive $B_p \times B_q$ $r \times r$ blocks. See Figure 4 for an example with $B_p = 4$ and $B_q = 3$: this panel of 12 $r \times r$ blocks will be distributed cyclically along both dimensions of the 2D grid. The previous cyclic dimension for homogeneous grids obviously corresponds to the case $B_p = p$ and $B_q = q$. Now, the distribution of individual blocks is no longer purely cyclic but remains periodic. We illustrate in Figure 5 how block panels are distributed on the 2D-grid.

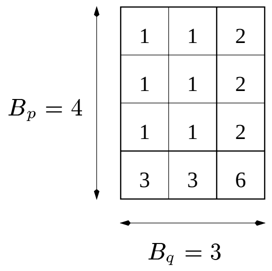

Figure 4: A block panel with $B_p = 4$ and $B_q = 3$. Each processor is labeled by its cycle-time, i.e the (normalized) time it needs to compute one $r \times r$ block: the processor labeled 1 is twice faster than the one labeled 2, hence it is assigned twice more blocks within each panel.

1	1	2	1	1	2	1	1	2	1
1	1	2	1	1	2	1	1	2	1
1	1	2	1	1	2	1	1	2	1
3	3	6	3	3	6	3	3	6	3
1	1	2	1	1	2	1	1	2	1
1	1	2	1	1	2	1	1	2	1
1	1	2	1	1	2	1	1	2	1
3	3	6	3	3	6	3	3	6	3
1	1	2	1	1	2	1	1	2	1
1	1	2	1	1	2	1	1	2	1

Figure 5: Allocating 4×3 panels on a 2×2 grid (processors are labeled by their cycle-time). There is a total of 10×10 matrix blocks.

How many $r \times r$ blocks should be assigned to each processor within a panel ? Intuitively, as in the case of uni-dimensional grids, the workload of each processor (i.e. the number of block per panel it is assigned to) should be inversely proportional to its cycle-time. In the example of Figure 4, we have a 2×2 grid of processors of respective cycle-time $t_{1,1} = 1$, $t_{1,2} = 2$, $t_{2,1} = 3$ and $t_{2,2} = 6$. The allocation of the $B_p \times B_q = 4 \times 3 = 12$ blocks of the panel perfectly balances the load amongst the four processors.

There is an important condition to enforce when assigning blocks to processors within a block panel. We want each processor in the grid to communicate only with its four direct neighbors. This implies that each processor in a grid row is assigned the same number of matrix rows. Similarly, each processor in a grid column must be assigned the same number of matrix columns. If these conditions do not hold, additional communications will be needed, as illustrated in Figure 6.

Translated in terms of $r \times r$ matrix blocks, the above conditions mean that each processor P_{ij}, $1 \le j \le q$ in the i-th grid row must receive the same number r_i of blocks. Similarly, P_{ij}, $1 \le i \le p$ must receive c_j blocks. This condition does hold in the example of Figure 5, hence each processor only communicates with its direct neighbors.

Unfortunately, and in contrast with the uni-dimensional case, the additional constraints induced by the communication pattern may well prevent to achieve a perfect load balance amongst processors. Coming back to Figure 4, we did achieve a perfect load balance, owing to the fact that the processor cycle-times could be arranged in the rank-1 matrix

$$\begin{pmatrix} t_{11} & t_{12} \\ t_{21} & t_{22} \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ 3 & 6 \end{pmatrix}.$$

For instance, change the cycle-time of $P_{2,2}$ into $t_{22} = 5$. If we keep the same allocation as in Figure 4, P_{22} remains idle every sixth time-step. Note that there is no solution to perfectly balance the work. Indeed, let r_1, r_2, c_1 and c_2 be the number of blocks assigned to each row and column grid. Processor P_{ij} computes $r_i \times c_j$ blocks in time $r_i \times c_j \times t_{ij}$. To have a perfect load balance, we have to fulfill the following equations:

$$r_1 \times t_{11} \times c_1 = r_1 \times t_{12} \times c_2 = r_2 \times t_{21} \times c_1 = r_2 \times t_{22} \times c_2$$

$$\text{that is} \quad r_1 c_1 = 2 r_1 c_2 = 3 r_2 c_1 = 6 r_2 c_2.$$

We derive $c_1 = 2c_2$, then $r_1 = 3r_2 = \frac{5}{2} r_2$, hence a contradiction. Note that we have not taken into account the additional condition $(r_1 + r_2) \times (c_1 + c_2) = 12$, stating that there are 12 blocks within a block panel: it is impossible to perfectly load-balance the work, whatever the size of the panel.

If we relax the constraints on the communication pattern, we can achieve a perfect load-balance as follows: first we balance the load in each processor column independently (using the uni-dimensional scheme); next we balance the load between columns (using the uni-dimensional scheme again, weighting each column by the inverse of the harmonic mean of the cycle-times of the processors within the column, see below). This is the "heterogeneous block cyclic distribution" of Kalinov and Lastovetky [20], which leads to the solution of Figure 6. Because processor $P_{2,2}$ has two west neighbors instead of one, at each step of the algorithm it is involved in two horizontal broadcasts instead of one.

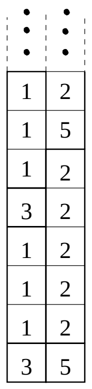

Figure 6: The distribution of Kalinov and Lastovetky. Two consecutive columns are represented here. Processors have two west neighbors instead of one.

We use the example to explain with further details how the heterogeneous block cyclic distribution of Kalinov and Lastovetky [20] works. First they balance the load in each processor column independently, using the uni-dimensional scheme. In the example there are two processors in the first grid column with cycle-times $t_{11} = 1$ and $t_{21} = 3$, so P_{11} should receive three times more matrix rows than P_{21}. Similarly for the second grid column, P_{12} (cycle-time $t_{12} = 2$) should receive 5 out of every 7 matrix rows, while P_{22} (cycle-time $t_{22} = 5$) should receive the remaining 2 rows. Next how to distribute matrix columns? The first grid column operates as a single processor of cycle-time $2\frac{1}{1+\frac{1}{3}} = \frac{3}{2}$. The second grid column operates as a single processor of cycle-time $2\frac{1}{\frac{1}{2}+\frac{1}{5}} = \frac{20}{7}$. So out of every 61 matrix columns we assign 40 to the first processor column and 21 to the second processor column.

Because we have a library designer's approach, we do not want the number of horizontal and vertical communications to depend upon the data distribution. For large grids, the number of horizontal neighbors of a given processor cannot be bounded a priori if we use Kalinov and Lastovetky's approach. We enforce the grid communication pattern (each processor only communicates with its four direct neighbors) to minimize communication overhead. The price to pay is that we have to solve a difficult optimization problem to load-balance the work as efficiently as possible. Solving this optimization problem is the objective of Section 4.

3.2 The LU and QR Decompositions

We first recall the ScaLAPACK algorithm for the LU or QR decompositions on a homogeneous 2D-grid. We discuss next how to implement them on a heterogeneous 2D-grid.

3.2.1 Homogeneous Grids

In this section we briefly review the direct parallelization of the right-looking variant of the LU decomposition. We assume that the matrix A is distributed onto a two-dimensional grid of (virtual) homogeneous processors. We use a `CYCLIC(b)` decomposition in both dimensions. The right-looking variant is naturally suited to parallelization and can be briefly described as follows: Consider a matrix A of order N and assume that the LU factorization of the $k \times b$ first columns has proceeded with $k \in \{0, 1, \ldots \frac{N-1}{b}\}$. During the next step, the algorithm factors the next panel of r columns, pivoting if necessary. Next the pivots are applied to the remainder of the matrix. The lower trapezoid factor just computed is broadcast to the other process columns of the grid using an increasing-ring topology, so that the the upper trapezoid factor can be updated via a triangular solve. This factor is then broadcast to the other process rows using a minimum spanning tree topology, so that the remainder of the matrix can be updated by a rank-r update. This process continues recursively with the updated matrix. In other words, at each step, the current panel of columns is factored into L and the trailing submatrix $\bar{A}$ is updated. The key computation is this latter rank-b update $\bar{A} \leftarrow \bar{A} - LU$ that can be implemented as follows:

1. The column processor that owns L broadcasts it horizontally (so there is a broadcast in each processor row)
2. The row processor that owns U broadcasts it vertically (so there is a broadcast in each processor column)
3. Each processor locally computes its portion of the update

The communication volume is thus reduced to the broadcast of the two row and column panels, and matrix $\bar{A}$ is updated in place (this is known as an outer -product parallelization). Load balance is very good. The simplicity of this parallelization, as well as its expected good performance, explains why the right-looking variants have been chosen in ScaLAPACK [9]. See [14, 9, 5] for a detailed performance analysis of the right-looking variants, that demonstrates their good scalability property. The parallelization of the QR decomposition is analogous [11, 10]

3.2.2 Heterogeneous Grids

For the implementation of the LU and QR decomposition algorithms on a heterogeneous 2D grid, we modify the ScaLAPACK $CYCLIC(r)$ distribution very similarly as for the matrix-matrix multiplication problem. The intuitive reason is the following: as pointed out before, the core of the LU and QR decompositions is a rank-r update, hence the techniques for the outer-product matrix algorithm naturally apply.

We still use block panels made up with several $r \times r$ matrix blocks. The block panels are distributed cyclically along both dimensions of the grid. The only modification if that the order of the blocks within a block panel becomes important.

Consider the previous example with four processors laid along a 2×2 grid as follows:

$$\begin{pmatrix} t_{11} & t_{12} \\ t_{21} & t_{22} \end{pmatrix} = \begin{pmatrix} 1 & 2 \\ 3 & 5 \end{pmatrix}.$$

Say we use a panel with $B_p = 8$ ad $B_q = 6$, i.e. a panel composed of 48 blocks. Using the methods described below (see Section 4), we assign he blocks as follows:

- Within each panel column, the first processor row receives 6 blocks and the second processor rows receives 2 blocks
- Out of the 6 panel columns, the first grid column receives 4 and the second grid column receive 2

This allocation is represented in Figure 7. We need to explain how we have allocated the six panel columns. For the matrix multiplication problem, the ordering of the blocks within the panel was not important, because all processors execute the same amount of (independent) computations at each step of the algorithm. For the LU and QR decomposition algorithms, the ordering of the columns is quite important: as explained in Section 2.3.2, the size of the matrix shrinks at each step of the computation. We use the uni-dimensional algorithm of Section 2.3.2 to compute the column ordering for the 2D panel. In the example, the first processor column operates like 6 processors of cycle-time 1 and 2 processors of cycle-time 3, which is equivalent to a single processor A of cycle-time $\frac{3}{20}$; the second processor column operates like 6 processors of cycle-time 2 and 2 processors of cycle-time 5, which is equivalent to a single processor B of cycle-time $\frac{5}{17}$. The uni-dimensional algorithm allocates the six panel columns as $ABAABA$, and we retrieve the allocation of Figure 7.

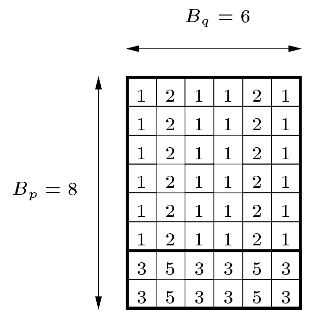

Figure 7: Allocation of the blocks within a block panel with $B_p = 8$ and $B_q = 6$. Each processor of the 2×2 grid is labeled by its cycle-time.

To conclude this section, we have a difficult load-balancing problem to solve. First we do not know which is the best layout of the processors, i.e. how to arrange them to build an efficient 2D grid. In some cases (rank-1 matrices) we are able to load-balance the work perfectly, but in most cases it is not the case. Next, once the grid is built, we have to determine the number of blocks that are assigned to each processor within a block panel. Again, this must be done so as to load-balance the work, because processors have different speeds. Finally, the panels are cyclically distributed along both grid dimensions. The rest of the paper is devoted to a solution to this difficult load-balancing problem.

4 Solving the 2D Heterogeneous Grid Allocation Problem

4.1 Problem Statement and Formulation

Consider n processors $P_1, P_2, \ldots, P_n$ of respective cycle-times $t_1, t_2, \ldots, t_n$. The problem is to arrange these processors along a two-dimensional grid of size $p \times q \leq n$, in order to compute the product $Z = XY$ of two $N \times N$ matrices as fast as possible. We need some notations to formally state this objective.

Consider a given arrangement of $p \times q \leq n$ processors along a two-dimensional grid of size $p \times q$. Let us re-number the processors as P_{ij}, with cycle-time t_{ij}, $1 \leq i \leq p, 1 \leq j \leq q$. Assume that processor P_{ij} is assigned a block of r_i rows and c_j columns of data elements, meaning that it is responsible for computing $r_i \times c_j$ elements of the Z matrix: see Figure 8 for an example.

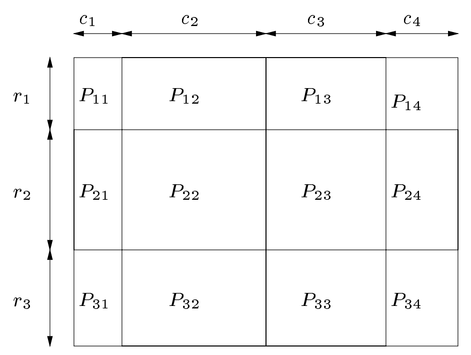

Figure 8: Allocating computations to processors on a 3×4 grid

There are two (equivalent) ways to compute the efficiency of the grid:

- Processor P_{ij} is assigned a rectangular data block of size $r_i \times c_j$, which it will process within $r_i \times c_j \times t_{ij}$ units of time. The total execution time T_{exe} is taken over all processors:

  $$T_{exe} = \max_{i,j}\{r_i \times t_{ij} \times c_j\}.$$

  T_{exe} must be normalized to the average time T_{ave} needed to process a single data element: since there is a total of N^2 elements to compute, we enforce that $\sum_{i=1}^{p} r_i = N$ and that $\sum_{j=1}^{q} c_j = N$. We get

  $$T_{ave} = \frac{\max_{i,j}\{r_i \times t_{ij} \times c_j\}}{\left(\sum_{i=1}^{p} r_i\right) \times \left(\sum_{j=1}^{q} c_j\right)}.$$

  We are looking for the minimum of this quantity over all possible integer values r_i and c_j. We can simplify the expression for T_{ave} by searching for (nonnegative) rational values r_i and c_j which sum up to 1 (instead of N):

  Objective Obj_1: $\min_{(\sum_{i=1}^{p} r_i = 1; \sum_{j=1}^{q} c_j = 1)}\{r_i \times t_{ij} \times c_j\}$

  Given the rational values r_i and c_j returned by the solution of the optimization problem Obj_1, we scale them by the factor N to get the final solution. We may have to round up some values, but we do so while preserving the relation $\sum_{i=1}^{p} r_i = \sum_{j=1}^{q} c_j = N$. Stating the problem as Opt_1 renders its solution generic, i.e. independent of the parameter N.

- Another way to tackle the problem is the following: what is the largest number of data elements that can be computed within one time unit? Assume again that each processor P_{ij} of the $p \times q$ grid is assigned a block of r_i rows and c_j columns of data elements. We need to have $r_i \times t_{ij} \times c_j \leq 1$ to ensure that P_{ij} can process its block within one cycle. Since the total number of data elements being processed is $\left(\sum_{i=1}^{p} r_i\right) \times \left(\sum_{j=1}^{q} c_j\right)$, we get the (equivalent) optimization problem:

  Objective Obj_2: $\max_{r_i \times t_{ij} \times c_j \leq 1}\{\left(\sum_{i=1}^{p} r_i\right) \times \left(\sum_{j=1}^{q} c_j\right)\}$

  Again, the rational values r_i and c_j returned by the solution of the optimization problem Obj_2 can be scaled and rounded to get the final solution.

  Although there are $p+q$ variables r_i and c_j, there are only $p+q-1$ degrees of freedom: if we multiply all r_i's by the same factor λ and divide all c_j by λ, nothing changes in Obj_2. In other words, we can impose $r_1 = 1$, for instance, without loss of generality.

We can further manipulate Obj_2 as follows:

$$\begin{aligned}
\max_{r_i \times t_{ij} \times c_j \leq 1}\{(\sum_{i=1}^{p} r_i) \times \left(\sum_{j=1}^{q} c_j\right)\} &= \max_{r_i}\{\max_{c_j \text{with} r_i \times t_{ij} \times c_j \leq 1}\{(\sum_{i=1}^{p} r_i) \times \left(\sum_{j=1}^{q} c_j\right)\}\} \\
&= \max_{r_i}\{(\sum_{i=1}^{p} r_i) \times \max_{c_j \text{with} r_i \times t_{ij} \times c_j \leq 1}\{\left(\sum_{j=1}^{q} c_j\right)\}\} \\
&= \max_{r_i}\{(\sum_{i=1}^{p} r_i) \times \max_{c_j \leq \frac{1}{r_i \times t_{ij}}}\{\left(\sum_{j=1}^{q} c_j\right)\}\} \\
&= \max_{r_i}\{(\sum_{i=1}^{p} r_i) \times \left(\sum_{j=1}^{q} \min_i\{\frac{1}{r_i \times t_{ij}}\}\right)\} \\
&= \max_{r_i}\{(\sum_{i=1}^{p} r_i) \times \left(\sum_{j=1}^{q} \frac{1}{\max_i\{r_i \times t_{ij}\}}\right)\}
\end{aligned}$$

We obtain an expression with only p variables (and $p-1$ degrees of freedom). This last expression does not look very friendly, though. Solving this optimization problem, optimally or through an heuristic, is the main objective of Section 4.3.

The 2D load-balancing problem In the next sections we give a solution to the 2D load-balancing problem which can be stated as follows: given $n = p \times q$ processors, how to arrange them along a 2D grid of size $p \times q$ so as to optimally load-balance the work of the processors for the matrix-matrix multiplication problem. Note that solving this problem will in fact lead to the solution of many linear algebra problems, including dense linear system solvers.

The problem is even more difficult to tackle than the optimization problem stated above, because we do not assume the processors arrangement as given. We search among all possible arrangements (layouts) of the $p \times q$ processors as a $p \times q$ grid, and for each arrangement we must solve the optimization problem Obj_1 or Obj_2.

We start with a useful result to reduce the number of arrangements to be searched. Next we derive an algorithm to solve the optimization problem Obj_1 or Obj_2 for a fixed (given) arrangement. Despite the reduction, we still have an exponential number of arrangements to search for. Even worse, for a fixed arrangement, our algorithm exhibits an exponential cost. Therefore we introduce a heuristic to give a fast but sub-optimal solution to the 2D load-balancing problem.

4.2 Reduction to Non-Decreasing Arrangements

The arrangement of the processors along the grid is a degree of freedom of the problem. For example when using a Myrinet network [13] we can define every desired topology for a fixed degree (number of neighbors) of the interconnection graph. Hence, finding a good arrangement is a key step of the load-balancing problem.

In this section, we show that we do not have to consider all the possible arrangements; instead, we reduce the search to "non-decreasing arrangements". A non-decreasing arrangement on a $p \times q$ grid is defined as follows: in every grid row, the cycle-times are increasing: $t_{ij} \leq t_{i,j+1}, 1 \leq j \leq q-1$. Similarly, in every grid column, the cycle-times are increasing: $t_{ij} \leq t_{i+1,j}, 1 \leq i \leq p-1$.

Theorem 1 *There exists a non-increasing arrangement which is optimal.*

Proof The proof works as follows:

1. Let the $p \times q$ cycle-times be denoted as $t_1, t_2, t_{p \times q}$
2. Consider an optimal arrangement of the grid. There is no reason that the optimal arrangement be a non-decreasing arrangement.
3. Show that some well-chosen "correct" transpositions can be applied to the arrangement while preserving the optimality of the solution. The correct transpositions will make the arrangement "closer" to a non-decreasing arrangement

We need a few definitions:

Definition 1 Arrangement *An arrangement is a one-to-one mapping*

$$\sigma : \left(\begin{array}{rcl} \{1, \ldots, p \times q\} & \longmapsto & \{1, \ldots, p\} \times \{1, \ldots, q\} \\ k & \longrightarrow & \sigma(k) = (i, j) \end{array} \right)$$

which assigns a position to each processor in the grid.

Non-decreasing arrangement *An arrangement σ is non-decreasing if $t_{\sigma^{-1}(i,j)} \leq t_{\sigma^{-1}(i',j')}$ for all $(1,1) \leq (i,j) \leq (i',j') \leq (p,q)$*

Correct transposition *Let σ be an arrangement. If $\sigma(k) < \sigma(l)$ and $t_k > t_l$, the transposition $\tau(k,l)$ which transposes the values of $\sigma(k)$ and $\sigma(l)$ is said to be correct.*

Given any arrangement σ, there exists a suite of of correct transpositions that modifies σ into a non-decreasing arrangement. To prove this, we use a *weight function* W that quantifies the distance to the "non-decreasing-ness", so to speak: a correct transposition will decrease the weight of the arrangement it is applied to. We search for

$$W : \left(\begin{array}{rcl} \Sigma_{p,q} & \rightarrow & \mathcal{N} \\ \sigma & \rightarrow & W(\sigma) \end{array} \right)$$

(where $\Sigma_{p,q}$ is the set of all arrangements and $\mathcal{N}$ represents the set of positive integers) such that for each correct transposition τ, $W(\tau(\sigma)) < W(\sigma)$. We choose

$$W(\sigma) = \sum_{i,j} t_{\sigma^{-1}(i,j)} \times (p + q - i - j)$$

To check that W has the desired property, let σ be an arrangement such that $(i,j) \leq (i',j')$ and $t_{\sigma^{-1}(i,j)} > t_{\sigma^{-1}(i',j')}$. Let $k = \sigma^{-1}(i,j)$ and $l = \sigma^{-1}(i',j')$: by hypothesis, the transposition $\tau = \tau(k,l)$ is correct. Let $\sigma' = \tau \circ \sigma$. We have

$$\begin{array}{rcl} W(\sigma') & = W(\sigma) & + \quad (t_{\sigma^{-1}(i',j')} - t_{\sigma^{-1}(i,j)})(p + q - i - j) \\ & & + \quad (t_{\sigma^{-1}(',j)} - t_{\sigma^{-1}(i',j')})(p + q - i' - j') \\ & = W(\sigma) & - \quad ((i' - i) + (j' - j))(t_{\sigma^{-1}(i,j)} - t_{\sigma^{-1}(i',j')}) \\ & < W(\sigma) & \end{array}$$

Consider an optimal arrangement σ, and let $r_1, \ldots, r_p$ and $c_1, \ldots, c_q$ be the solution to the optimization problem Obj_1. Since an equivalent solution is obtained by transposing two columns or two rows of the arrangement, we can assume that $r_1 \geq r_2 \geq \ldots r_p$ and $c_1 \geq c_2 \geq \ldots c_q$. If σ is non-decreasing, we are done. Otherwise, there exists $(1,1) \leq (i,j) < (p,q)$ such that either $t_{\sigma^{-1}(i+1,j)} < t_{\sigma^{-1}(i,j)}$ or $t_{\sigma^{-1}(i,j+1)} < t_{\sigma^{-1}(i,j)}$. The proof is the same in both cases, hence assume that $t_{\sigma^{-1}(i+1,j)} < t_{\sigma^{-1}(i,j)}$. Let $k = \sigma^{-1}(i,j)$ and $l = \sigma^{-1}(i+1,j)$: by hypothesis, the transposition $\tau = \tau(k,l)$ is correct. Let $\sigma' = \tau \circ \sigma$. We want to show that σ' is as good as σ, which achieves the quantity $\mathcal{O} = (\sum r_i \times \sum c_j$ subject to $\max_{i,j}(r_i c_j t_{\sigma^{-1}(i,j)}) \leq 1$. We have $r_i \geq r_{i+1}$ and $t_{\sigma^{-1}(i+1,j)} < t_{\sigma^{-1}(i,j)}$, hence

$$\left\{ \begin{array}{l} r_i c_j t_{(\tau \circ \sigma)^{-1}(i,j)} = r_i c_j t_{\sigma^{-1}(i+1,j)} \leq r_i c_j t_{\sigma^{-1}(i,j)} \leq 1 \\ r_{i+1} c_j t_{(\tau \circ \sigma)^{-1}(i+1,j)} = r_{i+1} c_j t_{\sigma^{-1}(i,j)} \leq r_i c_j t_{\sigma^{-1}(i,j)} \leq 1 \end{array} \right.$$

Therefore, σ' is optimal too, and $W(\sigma') < W(\sigma)$. If σ' is not non-decreasing, we repeat the process, which converges in a finite number of steps, because there is a finite number of weight values. ∎

4.3 Solution for a Given Arrangement

In this section, we show how to solve the optimization problem Obj_1 or Obj_2 for a given arrangement. For small size problems, all the possible non-decreasing arrangements can be generated, hence we have an exponential but feasible solution to the 2D load balancing problem. Let σ be a given arrangement on a $p \times q$ grid, and let $r_1, \ldots, r_p$ and $c_1, \ldots, c_q$ be the solution to the optimization problem Obj_1

4.3.1 Spanning Trees

Consider the optimization problem Obj_1. We have to maximize the quadratic expression $(\sum_{1 \leq i \leq p} r_i)(\sum_{1 \leq j \leq q} c_j)$ under $p \times q$ inequalities $r_i t_{ij} c_j \leq 1$. We have $p+q-1$ degrees of freedom. The objective of this section is to show that for at least $p+q-1$ inequalities are in fact equalities. We use a graph-oriented approach to this purpose.

We consider the following bipartite graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. There are $p+q$ vertices labeled with r_i and c_j and the graph is complete. The weight of the edge (r_i, c_j) is t_{ij}. Given a spanning tree $\mathcal{T} = (\mathcal{V}, \mathcal{E}')$ of the graph $\mathcal{G}$, if we start from $r_1 = 1$, we can (uniquely) determine all the values of the r_i and c_j by following the edges of $\mathcal{T}$, enforcing the equalities

$$\forall (r_i, c_j) \in \mathcal{E}', \;\; r_i t_{i,j} c_j = 1.$$

The spanning tree $\mathcal{T}$ is said to be *acceptable* if and only if all the remaining inequalities are satisfied: $\forall (r_i, c_j) \in \mathcal{E}, \;\; r_i t_{i,j} c_j = 1.$. The value of an acceptable spanning tree is $(\sum r_i)(\sum c_j)$. We claim that the solution of Obj_1 is obtained with the acceptable spanning tree of maximal value. This leads to the following algorithm:

Algorithm We generate all the spanning trees of $\mathcal{G}$. For a given tree $\mathcal{T}$, we first impose that $r_1 = 1$, then by walking on the tree we find the values for the other r_i and c_j. For example, if c_3 is connected to r_1 in $\mathcal{T}$, then we take $c_3 = \frac{1}{r_1 t_{13}}$. When we have a value for all r_i and c_j, we check if the tree is acceptable. Finally, we select the acceptable tree that maximizes $\sum_{1 \leq i \leq p} r_i \sum_{1 \leq j \leq q} c_j$.

Justification To justify the previous algorithm, consider an optimal solution to Obj_1 and draw the bipartite graph $\mathcal{U} = (\mathcal{V}, \mathcal{E}')$ corresponding to the equalities: $\mathcal{U}$ has $p+q$ vertices labeled with r_i and c_j. There is an edge between vertices r_i and c_j ($(r_i, c_j) \in \mathcal{E}'$) iff $r_i c_j t_{i,j} = 1$. If $\mathcal{U}$ is connected, we are done. Otherwise, we decrease the number of connected components in $\mathcal{U}$, assigning new values to some r_i and c_j, but without decreasing the quantity $\sum_{1 \leq i \leq p} r_i \sum_{1 \leq j \leq q} c_j$. Let $\mathcal{V}'$ be a connected component of $\mathcal{U}$. Suppose (without any loss of generality) that $\mathcal{V}' = \{r_1, \ldots, r_{p'}, c_1, \ldots, c_{q'}\}$ with $p' < p$.

First case If $q' = q$, it means that for all $1 \leq j \leq q$, $r_{p'+1} c_j t_{p'+1,j} < 1$. Let $\alpha = \min_{1 \leq j \leq q} \frac{1}{r_{p'+1} c_j t_{p'+1,j}}$. Then, $r_{p'+1}$ can be increased by a factor of α, and the product $\sum_{1 \leq i \leq p} r_i \sum_{1 \leq j \leq q} c_j$ is increased too. Moreover, the vertex $r_{p'+1}$ becomes connected to the component $\mathcal{V}'$. So the number of connected components is decreased by one.

Second case If $q' < q$, it means that for all $(p'+1, 1) \leq (i,j) \leq (p, q')$ and for all $(1, q'+1) \leq (i,j) \leq (p', q)$, $r_i c_j t_{i,j} < 1$. Let $\alpha_r = \min_{(1,q'+1) \leq (i,j) \leq (p',q)} \frac{1}{r_i c_j t_{i,j}}$ and $\alpha_c = \min_{(p'+1,1) \leq (i,j) \leq (p,q')} \frac{1}{r_i c_j t_{i,j}}$. We also introduce the notations $R_a = \sum_{1 \leq i \leq p'} r_i$, $R_b = \sum_{p'+1 \leq i \leq p} r_i$, $C_a = \sum_{1 \leq j \leq q'} c_j$ and $C_b = \sum_{q'+1 \leq j \leq q} c_j$.

Now, we have two possibilities to increase the connectivity of the graph: either increase R_b and decrease C_b by a factor of α_r or decrease R_b and increase C_b by a factor of α_c. We must check that at least one of this solution does increase the product $(R_a + R_b)(C_a + C_b)$. Indeed, consider the function

$$f : \left(\begin{array}{ccc} \mathcal{R}_+^* & \longmapsto & \mathcal{R}_+^* \\ \lambda & \longrightarrow & f(\lambda) = (\lambda R_b + R_a)(\frac{C_b}{\lambda} + C_a) \end{array} \right)$$

Note that $f'(1) = R_b C_a - R_a C_b$. Moreover, f is a continuous function that is first decreasing to a minimum and then increasing. Therefore,

- if $f'(1) \geq 0$, then for all $\lambda \geq 1$, $f(\lambda) \geq f(1)$. In particular, $f(\alpha_r) > f(1)$.
- if $f'(1) \leq 0$, then for all $\lambda \leq 1$, $f(\lambda) \geq f(1)$. In particular, $f(\frac{1}{\alpha_c}) > f(1)$.

One of the previous two solutions will indeed increase the connectivity of $\mathcal{U}$ while preserving the objective function $\sum_{1 \leq i \leq p} r_i \sum_{1 \leq j \leq q} c_j$. We conclude that there does exist an acceptable spanning tree whos value is the optimal solution.

To summarize, given an arrangement, we are able to compute the solution to the optimization problem. The cost is exponential because there is an exponential number of spanning trees to check for acceptability. Still, our method is constructive, and can be used for problems of limited size.

In the rest of this section we deal with two particular simple cases.

4.3.2 Case of a 2×2 Grid

For small size grids, we can find analytical solutions to the optimization problem. As an example, we will solve in this section the 2×2 case.

In the 2×2 case, we want to maximize (see the definition of Obj_2) the quantity $(r_1 + r_2)(\frac{1}{\max(r_1 t_{11}, r_2 t_{21})} + \frac{1}{\max(r_1 t_{12}, r_2 t_{22})})$. We normalize our problem by letting $r_1 = 1$ and $r_2 = r$. We have to maximize

$$(1+r)(\frac{1}{max(t_{11}, rt_{21})} + \frac{1}{max(t_{12}, rt_{22})}).$$

There are three cases to study:

First case $0 \leq r \leq min(\frac{t_{11}}{t_{21}}, \frac{t_{12}}{t_{22}})$: in this interval, the value of the expression varies as an increasing function of r. So the maximum on this interval is obtained for $r = min(\frac{t_{11}}{t_{21}}, \frac{t_{12}}{t_{22}})$.

Second case $max(\frac{t_{11}}{t_{21}}, \frac{t_{12}}{t_{22}}) \leq r \leq +\infty$: in this interval, the expression varies as a decreasing function of r. So, the maximum is obtained for $r = max(\frac{t_{11}}{t_{21}}, \frac{t_{12}}{t_{22}})$.

Third case $min(\frac{t_{11}}{t_{21}}, \frac{t_{12}}{t_{22}}) \leq r \leq max(\frac{t_{11}}{t_{21}}, \frac{t_{12}}{t_{22}})$: by symmetry, we can suppose that $\frac{t_{11}}{t_{21}} \leq \frac{t_{12}}{t_{22}}$. So our expression is now $(1+r)(\frac{1}{rt_{21}} + \frac{1}{t_{12}})$. This function is first decreasing and then increasing. Hence, the maximum is obtained on the bounds of the interval, i.e. for $r = \frac{t_{11}}{t_{21}}$ or $r = \frac{t_{12}}{t_{22}}$.

In conclusion, there are 2 possible values for the maximum, namely $r = \frac{t_{11}}{t_{21}}$ or $r = \frac{t_{12}}{t_{22}}$. For the objective function we obtain the value

$$\max\left(\left(1 + \frac{t_{11}}{t_{21}}\right)\left(\frac{1}{t_{11}} + \frac{1}{\max(t_{12}, \frac{t_{11}t_{22}}{t_{21}})}\right), \left(1 + \frac{t_{12}}{t_{22}}\right)\left(\frac{1}{\max(t_{11}, \frac{t_{12}t_{21}}{t_{22}})} + \frac{1}{t_{12}}\right)\right)$$

4.3.3 Rank-1 Matrices

If the matrix $(t_{ij})_{1 \leq i \leq p, 1 \leq j \leq q}$ is a rank-1 matrix, then the optimal arrangement for the 2D load-balancing problem is easy to determine. Assume without loss of generality that $t_{11} = 1$. We let $r_1 = c_1 = 1$, $r_i = \frac{1}{t_{i1}}$ for $2 \leq i \leq p$ and $c_j = \frac{1}{t_{1j}}$ for $2 \leq i \leq q$. All the $p \times q$ inequalities $r_i t_{ij} c_j$ are equalities, which means that all processors are fully utilized:

$$r_i t_{ij} c_j = \frac{1}{t_{i1}} t_{ij} \times \frac{1}{t_{j1}} t_{ij} = 1,$$

because the 2×2 determinant $\begin{vmatrix} t_{11} & t_{1j} \\ t_{i1} & t_{ij} \end{vmatrix}$ is zero (with $t_{11} = 1$). No idle time occurs with such a solution, the load-balancing is perfect.

Unfortunately, given $p \times q$ integers, it is very difficult to know whether they can be arranged into a rank-1 matrix of size $p \times q$. If such an arrangement do not exist, we can intuitively say that the optimal arrangement is the "closest one" to a rank-1 matrix.

4.4 Polynomial Heuristic

In this section, we study a polynomial heuristic to find an arrangement of the processors (and a solution to the corresponding optimization problem) that leads to a good load-balancing.

We are given the processor cycle-times as input to the heuristic. Either all cycle-times are relatively close, or there are some very slow machines whose cycle-times are much larger than the others. Such slow machines should be either discarded or, at least, grouped in some grid row or column so that they do not interfere with the faster processors.

In other words we want to separate the case where some processors are very slow compared to the others, from the case where all processor speeds are relatively close. We use a probabilistic criteria to this purpose: consider $p \times q$ integers $x_1, x_2, \cdots, x_{pq}$ increasingly sorted ($x_1 \leq x_2 \leq \cdots \leq x_{pq}$). Let e_k be the average of the first k integers and σ_k the standard deviation of these k integers: $\sigma_k = (\sum_{i=1}^{k}(x_i - e_k))^{\frac{1}{2}}$. We study the graph of $\frac{\sigma_k}{k}$: if there exists a jump for $k = k_0$ in the graph, we will consider that the x_i with $i > k_0$ are larger than the others. Otherwise, if there is no jump, we will consider that all the values are close. We detect this jump by measuring the angle between two pairs of consecutive points. If this angle is greater than a fixed value, we say there is a jump. Consider the following example: $X = (1, 2, 3, 4, 5, 6, 7, 100, 100)$. We see on Figure 9 that there is a jump before the last two integers. Consequently, we separate them from the others.

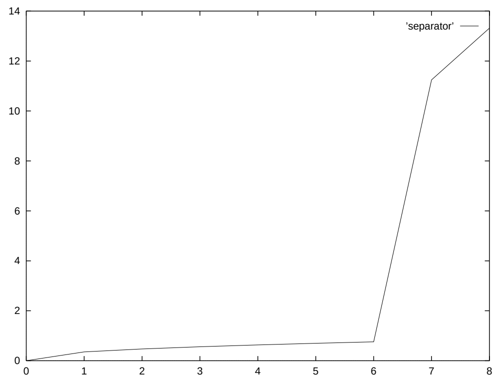

Figure 9: Graph of $\frac{\sigma_k}{k}$

In the following we study the two different cases when mapping $p \times q$ processors P_k of cycle-time t_k on a $p \times q$ grid:

4.4.1 First Case: All Values Are Close

First we sort all the values. Then we put the first $p + q - 1$ values on the border of the grid (first row and column). We would like to balance the values in th row and in the column, which means $\sum_{i=1}^{p} t_{1i} = \sum_{j=1}^{q} t_{1j}$. Because this PARTITION problem is NP-complete [18], we simply put alternatively the t_{ij} on the first row and on the first column of the grid.

Consider the following example for $p = q = 5$ with $t_k = k, 1 \leq k \leq 25$. Let $\mathcal{A}_h$ be the arrangement given by our heuristic. We get for the first nine values:

$$\mathcal{A}_h = \begin{pmatrix} 1 & 3 & 5 & 7 & 9 \\ 2 & & & & \\ 4 & & & & \\ 6 & & & & \\ 8 & & & & \end{pmatrix}$$

Then we continue the same strategy with the other values by placing them on the remaining grid, on the second row and column, next on the third row and column, and so on. The final solutions is:

$$\mathcal{A}_h = \begin{pmatrix} 1 & 3 & 5 & 7 & 9 \\ 2 & 10 & 12 & 14 & 16 \\ 4 & 11 & 17 & 19 & 21 \\ 6 & 13 & 18 & 22 & 24 \\ 8 & 15 & 20 & 23 & 25 \end{pmatrix}$$

4.4.2 Second Case: There Are Some Larger Values

With the previous criteria, we decompose the set $\mathcal{C}$ of the processor cycle-times into two subsets $\mathcal{C}_1$ and $\mathcal{C}_2$ where the cycle-times in $\mathcal{C}_2$ are greater than in $\mathcal{C}_1$. If we consider the example $\mathcal{C} = (1, 2, 3, 4, 5, 6, 7, 100, 100)$, we have $\mathcal{C}_1 = (1, 2, 3, 4, 5, 6, 7)$ and $\mathcal{C}_2 = (100, 100)$.

Although some processors are very slow, we want to use them, but we put them on the same grid row (or on the same grid column, or on both a row and a column, according to the number of slow processors).

The intuition behind this is to prevent them from slowing the other processors. If necessary, we fill the row made up with the slow processors with the slowest numbers of $\mathcal{C}_1$. In our example we put $(100, 100, 7)$ on a same column.

Then, we fill the remaining grid with the remaining processors as in the previous case. In the example we obtain:

$$\mathcal{A}_{heur} = \begin{pmatrix} 1 & 3 & 7 \\ 2 & 5 & 100 \\ 4 & 6 & 100 \end{pmatrix}$$

4.4.3 Computing the r_i and the c_j

Once the arrangement has been found, we have to find values for the r_i and c_j. Because the spanning tree method has a cost exponential in $p+q$, we use a polynomial heuristic instead. We consider the first row or the first column in the submatrix of the faster processors, depending upon which one is faster. We evaluate the speed of a row or a column by computing the harmonic mean of the cycle-times. The speed of n processors is defined by $n \times \frac{1}{\sum_{i=1}^{n} \frac{1}{t_i}}$.

For example, suppose that the first column is faster. We impose the values for the r_i: we let $r_i = \frac{1}{t_{i1}}$ for $1 \leq i \leq p$. Now, to define the remaining c_j, we use the conditions $r_i t_{ij} c_j \leq 1$, hence $c_j = \frac{1}{\max_i r_i t_{ij}}$, where i belongs to the submatrix. If there are some slow processors, we first compute the r_i and the c_j corresponding to the submatrix of fast processors; then we compute the remaining r_i or c_j at the end.

For instance, consider the following arrangement:

$$\mathcal{A} = \begin{pmatrix} 1 & 3 & 5 \\ 2 & 6 & 7 \\ 4 & 8 & 9 \end{pmatrix}$$

We will first consider the processors of cycle-times $1, 2, 4$ in the first column because they are faster than those of cycle-times $1, 3, 5$ in the first row. Indeed, the average speed of the first column is $3 \times \frac{1}{1+\frac{1}{2}+\frac{1}{4}} = 1.7$ while the average speed of the first row is $3 \times \frac{1}{1+\frac{1}{3}+\frac{1}{5}} = 1.95$. So we first compute the r_i. We have $r_1 = 1$, $r_2 = \frac{1}{2}$ and $r_3 = \frac{1}{4}$. Then the constraints impose $c_1 = \frac{1}{\max_i r_i t_{i1}} = 1$, $c_2 = \frac{2}{\max_i r_i t_{i2}} = \frac{1}{3}$, $c_3 = \frac{3}{\max_i r_i t_{i3}} = \frac{1}{5}$.

Here is an example with slow processors:

$$\mathcal{A} = \begin{pmatrix} 1 & 4 & 8 \\ 1 & 6 & 8 \\ 1 & 8 & 8 \end{pmatrix}$$

There are 6 slow processors, which we assign to the second and third grid columns. The submatrix of fast processor is simply the first grid column We compare the speed of $1, 1, 1$ with the speed of 1. The two speeds are equals, so we first compute the r_i because the length of the first column is greater than the length of the first row. Then we compute the c_j. We find so $r_1 = 1$, $r_2 = 1$, $r_3 = 1$, $c_1 = \frac{1}{\max(r_i t_{i1})} = 1$, $c_2 = \frac{1}{\max(r_i t_{i2})} = \frac{1}{8}$ and finally $c_3 = \frac{1}{\max(r_i t_{i3})} = \frac{1}{8}$.

4.4.4 Refinement

Consider the Obj_1 optimization problem of Section 4.1. In this formulation, values of r_i and c_j can be separated: with fixed values of r_i we easily find the best values of c_j, and reciprocally. We use this idea to implement a cheap refinement algorithm: to improve a given solution r_i, c_j, we alternatively: fix r_i, calculate the corresponding best c_j; fix c_j, calculate the corresponding best r_i, and so on. It can be written as follow:

```
$last = +\infty$
$fixed = r$
**while** $(\max_{i,j}(r_i c_j t_{i,j}) < last)$
```

```
  $last = \max_{i,j}(r_i c_j t_{i,j})$
  if $(fixed == r)$
    $k_i = \max_j(c_j t_{i,j})$
    $r_i = \frac{\frac{1}{k_i}}{\sum_l \frac{1}{k_l}}$
    $fixed = c$
  else
    $k_j = \max_i(r_i t_{i,j})$
    $c_j = \frac{\frac{1}{k_j}}{\sum_l \frac{1}{k_l}}$
    $fixed = r$
  end
end
```

This refinement is used to improve the solution given in the previous section.

5 MPI Experiments

We report several experiments in this section. First, we present the HNOW used for those experiments. Then, we present the different arrangements and allocations that we have tested. Finally, we report the results of the experiments, which we compare to the standard ScaLAPACK block-cyclic distribution.

5.1 Description of the HNOW

We use 9 heterogeneous Sun workstations whose characteristics are summarized in Table 3. These 9 processors are linked by a local Ethernet network. Hence, as already said in section 2, all communications are performed mostly sequentially.

Name	Description	Execution time t_i
smirnoff	SS 5	7.8
guinness	Ultra 1	1.0
farot	Ultra 1	1.0
arquebuse	SS 20	4.0
zelfde	Ultra 1	1.0
loop	SS 5	6.3
isostar	SS 5	7.8
arnica	SS 5	7.95
utltrafuel	SS 5	8

Table 3: Description of the processors. The (relative) cycle time t_i have been determined by running the matrix-matrix multiplication routine on each processor, with matrices of size 500×500.

5.2 Description of the Grids

With 9 processors, there are several possible grids: 3×3, 2×4, 1×9, 4×2. Because we concentrate on 2D-grids, we limit our experiments to the 3×3 and 2×4 grids (likely to perform better than a 4×2 grid for LU decomposition, due to pivoting issues).

5.2.1 Grid of size 3×3

We have 9 processors of relative cycle time $t_i \in \{1.0, 1.0, 1.0, 4.0, 6.3, 7.8, 7.8, 7.95, 8.0\}$. The polynomial heuristic of Section 4.4, in that case, separates processors of cycle time $\{1.0, 1.0, 1.0\}$ from the others. Hence, we obtain the following arrangement

$$\mathcal{A} = \begin{pmatrix} 1.0 & 4.0 & 7.8 \\ 1.0 & 6.3 & 7.95 \\ 1.0 & 7.8 & 8.0 \end{pmatrix}$$

The heuristic forces the faster processors to never remain idle: we obtain $c_1 = 1$ and $r_1 = r_2 = r_3 = 1$. Then, the other values can be easily determined: we get, $c_2 = 0.128$ and $c_3 = 0.125$.

For small size problems, it is easy to generate all possible solutions and then to derive the optimal one. We have implemented the (exponential) algorithms given in Sections 4.2 and 4.3. It turns out that the solution given by the heuristic is indeed optimal.

5.2.2 Grid of size 2×4

In this case, we keep the fastest 8 processors. As for the previous grid, the first 3 processors are separated from the others. Then the first row is completed with the fourth fastest processor. Finally, we obtain the following arrangement

$$\mathcal{A} = \begin{pmatrix} 1.0 & 1.0 & 1.0 & 4.0 \\ 6.3 & 7.8 & 7.8 & 7,95 \end{pmatrix}$$

Again, the heuristic forces the load of the 4 fast processors to be maximum. We get $r_1 = c_1 = c_2 = c_3 = 1$ and $c_4 = 0.25$. The value of the last variable follows: $r_2 = 0.128$. As for the previous case, the solution obtained by the heuristic turns out to be optimal.

5.3 Results

We compare the standard ScaLAPACK cyclic allocation ($CYCLIC(r,r)$ to be precise) with an MPI implementation of our static distribution, using a single panel of size $B_p = B_q = \lceil \frac{n}{r} \rceil$ for $n \times n$ matrices. We let $r = 64$ in all experiments.

Assessing the Experiments To assess the relevance of our results, we have to investigate the value of a "reasonable" (theoretical) speedup that should be targeted. For a given solution of the Opt_2 problem, the value of $\mathcal{W} = (\sum_i r_i) \times (\sum_j r_i)$ represents the (normalized) total amount of work executed in one time unit. Hence, for the 2 previous grids, given the values of the r_i and c_j, we get

$$\mathcal{W}_{hetero(3\times3)} = 3.76 \text{ and } \mathcal{W}_{hetero(2\times4)} = 3.67.$$

To calculate $\mathcal{W}$ for a purely cyclic allocation, we set $r_i = r_{i'}$ and $c_j = c_{j'}$ for all i, i', j and j'. Hence for the 3×3 grid, we get $r_i = 1$ and $c_j = \frac{1}{8}$, hence $\mathcal{W}_{cyclic(3\times3)} = 1.125$. For the 2×4 grid, we get $\mathcal{W}_{cyclic(2\times4)} = 1.01$.

Consequently, the best speedup that can be achieved is $\mathcal{S}_{3\times3} = \frac{\mathcal{W}_{hetero(3\times3)}}{\mathcal{W}_{cyclic(3\times3)}} = 3.34$ for the 3×3 grid, and $\mathcal{S}_{2\times4} = \frac{\mathcal{W}_{hetero(2\times4)}}{\mathcal{W}_{cyclic(2\times4)}} = 3.63$ for the 2×4 grid.

Experiments For each algorithm (MM, LU and QR) and for both grids, we represent on the same graph

- the actual execution time for a **cyclic(64,64) distribution**: blocks are distributed in a cyclic manner over the processors within each dimension. This is the ScaLAPACK distribution.
- the actual execution time for the **hetero(64,64) distribution**: blocks are distributed using the algorithm described in Section 3.2.2.
- the theoretical execution time of an algorithm that would achieve the best theoretical speedup $\mathcal{S}_{3\times3} = 3.34$ and $\mathcal{S}_{2\times4} = 3.63$. We simply multiply the actual execution time for the ScaLAPACK distribution by this speedup.

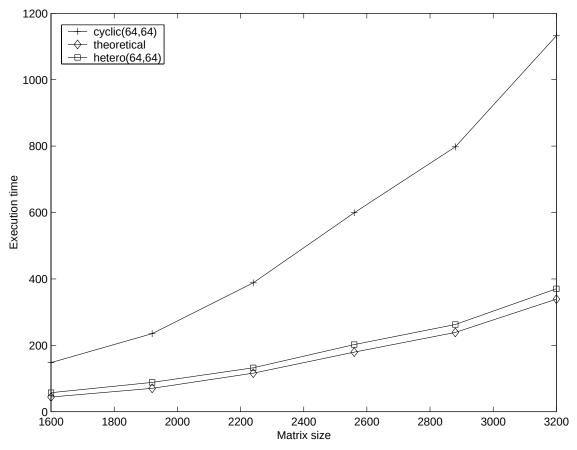

Figure 10: Matrix multiplication on a 3×3 grid.

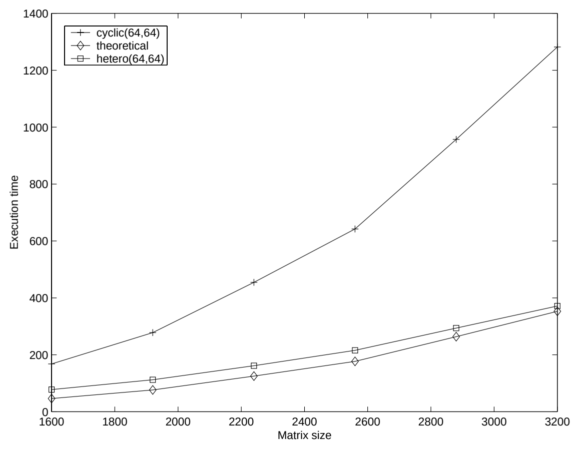

Figure 11: Matrix multiplication on a 2×4 grid.

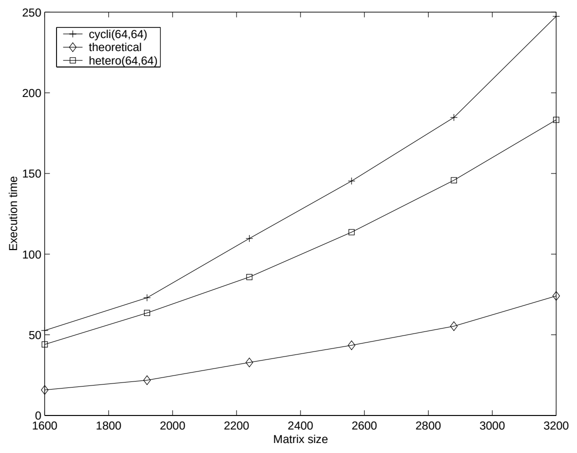

Figure 12: LU decomposition on a 3×3 grid.

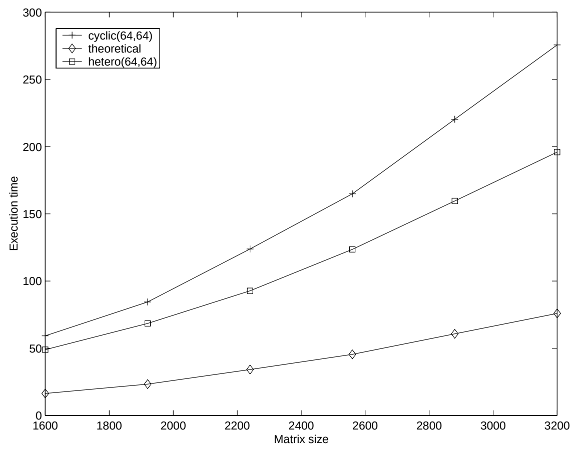

Figure 13: LU decomposition on a 2×4 grid.

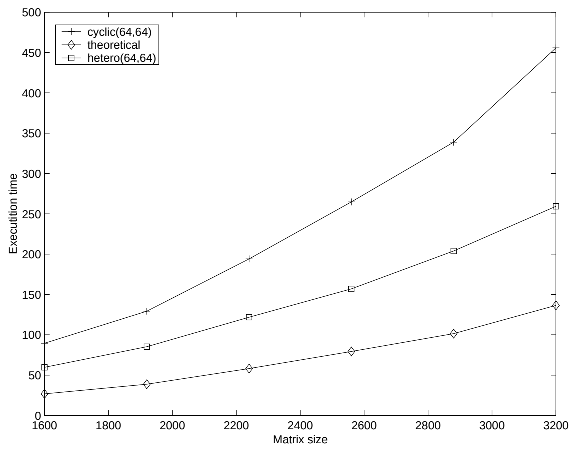

Figure 14: QR decomposition on a 3×3 grid.

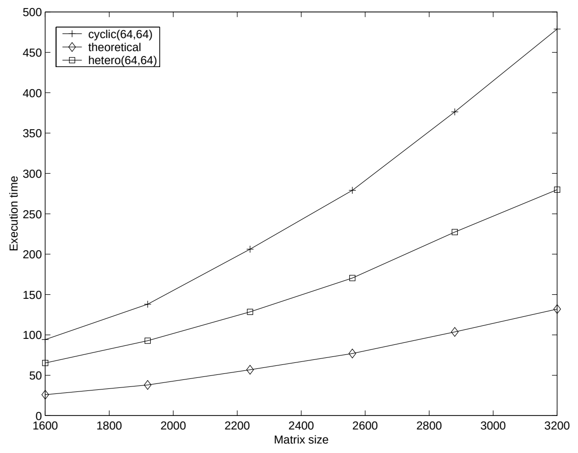

Figure 15: QR decomposition on a 2×4 grid.

The graphs of Figures 10 and 11 nicely demonstrate the relevance of our heterogeneous distribution strategy for the matrix multiplication problem. The actual performance optained for both grids is very close to the theorical speed-up.

For LU and QR decomposition, the results are not convincing. Still, we do observe (i) an important gain in the execution time which is more than halved ; (ii) a steady increase of this gain as the matrix size increases. We stress that the theorical speedup does not take into account neither the impact of the communications nor that of the pivoting, both of them being executed less efficiently on the slower processors. To summarize, these results are very encouraging because they demonstrate that a significant improvement can be made out of a very unbalanced HNOW as compared to the current ScaLAPACK implementation. However, there is room to improve further our solution (a possibility would be to have the fastest processor column execute all the pivoting steps).

6 Conclusion

In this paper, we have discussed static allocation strategies to implement matrix-matrix products and dense linear system solvers on heterogeneous computing platforms. Extending the standard ScaLAPACK block-cyclic distribution to heterogeneous 2D grids turns out to be surprisingly difficult. In most cases, a perfect balancing of the load between all processors cannot be achieved, and deciding how to arrange the processors along the 2D grid is a challenging problem. But we have formally stated the optimization problem to be solved, and we have presented both an exact solution (with exponential cost) and a polynomial heuristic.

Preliminary MPI experiments run a heterogeneous network fully demonstrate the practical value of our static allocation strategy. We believe to have provided an important contribution to the design of a complete ScaLAPACK library for heterogeneous clusters.

7 Appendix : description of the algorithms

7.1 Generation of the arrangements

Generating non-decreasing arrangements is equivalent to a string matching problem. To help understanding, let's introduce a few notations: let $\Sigma = \{1, 2, \ldots, p \times q\}^*$ (the set of strings of the alphabet $\{1, 2, \ldots, p \times q\}$). Let uv be the *concatenation* of the strings u and v (if $u = u_1u_2 \ldots u_n$ & $v = v_1v_2 \ldots v_m$, then $uv = u_1u_2 \ldots u_nv_1v_2 \ldots v_m$). Hence, if u and v have a respective *length* of n and m (which can be denoted by $|u| = n$ and $|v| = m$), then $|uv| = n+m$. We also say that u is a *prefix* of length n of the string uv. Finally, we define the *weight* of a character α in the string $u = u_1u_2 \ldots u_n$ as the integer $|u|_\alpha = \mathbf{Card}\{i \in \{1, \ldots, n\}, u_i = \alpha\}$.

Now, we define a *well-balanced string* of length n to be a string u such that

- for all prefix v of u, for all character $i \in \{2, \ldots, p \times q\}$, $|v|_{i-1} \geq |v|_i$.
- for all characters $(i, j) \in \{1, \ldots, p \times q\}^2$, $|u|_i = |u|_j$.

There is an isomorphism between the non-decreasing arrangements for a grid of size $p \times q$, and the well-balanced strings of length $p \times q$ on an alphabet of size q. The following algorithm generates very efficiently all the non-decreasing arrangements by taking this remark into account. It needs the following asumption: $[\forall (i, j) \in \{1, \ldots, p \times q\}^2, i \leq j \Longrightarrow t_i \leq t_j]$.

```
global variables :    $deep$, $\sigma$, $a_{\{1,\ldots,q\}}$
procedure generate_arrangement_rec()
```

```
  local variables :   $i$
  $deep \leftarrow deep + 1$
  if $(deep = p + q)$
    $\sigma(p, q) \leftarrow deep$
    $\sigma$ is a correct arrangement
  else
    forall $i \in \{1, \ldots, p\}$, $(i = 1$ or $a_{i-1} > a_i)$ and $a_i < q$
      $a_i \leftarrow a_i + 1$
      $\sigma(i, a_i) \leftarrow deep$
      generate_arrangement_rec()
      $a_i \leftarrow a_i - 1$
    endforall
  endif
  $deep \leftarrow deep - 1$
endprocedure
```

```
procedure generate_arrangement()
  forall $i \in \{1, \ldots, p\}$, $a_i \leftarrow 0$
  $deep \leftarrow 0$
  generate_arrangement_rec()
endprocedure
```

To evaluate the complexity of this algorithm, we remark that the number of well-balanced strings of length $p \times q$ on an alphabet of size q is the number of linear extensions of a partial order defined by a rectangle of size $p \times q$. Hence, we get from the Hook formula [17] a complexity of

$$\frac{(pq)! \prod_{i=1}^{p-1} i!}{\prod_{i=q}^{q+p-1} i!}$$

p & q	1	2	3	4	5
1	1	1	1	1	1
2	1	2	5	14	42
3	1	5	42	462	6006
4	1	14	462	24024	1662804
5	1	42	6006	1662804	701149020

Table 4: Number of non-decreasing arrangements.

7.2 Generation of the spanning trees

Generating all the subsets of a given set To generate all the spanning trees of a bipartite graph, we have to generate efficiently all the subsets of a given set. The method is simple, it uses the arithmetic incrementation algorithms ; one must previously make the assumption that E is a totaly ordered set. Then, the following loop generates all the subsets F of E:

```
$F = \emptyset$
do
  $e = max\{e \in E, e \notin F\}$
  $F \leftarrow F \cup \{e\} \setminus \{f \in F, f > e\}$
until $(F = E)$
```

Generating all the spanning trees of a bipartite graph The method consists on taking the vertex r_1 has the root. Then we build the tree, depth by depth:

```
Global variables :   $v$
procedure Generate_tree
  $v(r_1) = 1$
  forall $(i,j) \in \{1,\ldots,p\} \times \{1,\ldots,q\}$, $v((r_i,c_j)) = t_{i,j}$
  Generate_tree_rec($\emptyset, \{r_1\}, \{r_2, r_3, \ldots, r_p\}, \emptyset, \{c_1, c_2, \ldots, c_q\}$)
endprocedure


procedure Generate_tree_rec($R_0, R_1, R_2, C_0, C_2$)
  forall $c \in C_2$
    $v(c) = \min_{r \in R_0 \cup R_1} \left( \frac{1}{v(r) \times v((r,c))} \right)$
    $\alpha(c) = \left\{ r \in R_0 \cup R_1,\, v(c) = \frac{1}{v(r) \times v((r,c))} \right\}$
  endforall
  $C_1 \leftarrow \{c \in C_2,\, \alpha(c) \cap R_0 = \emptyset\}$
  if ($R_2 \neq \emptyset$)
    forall $C_1' \in \mathcal{P}(C_1) \setminus \{\emptyset\}$, Generate_tree_rec($C_0, C_1', C_2 - C_1', R_0 \cup R_1, R_2$)
  else if ($C_1 = C_2$)
    $v$ is an acceptable solution
  endif
endprocedure
```

Hence, the complexity ($Comp(p,q)$) of this algorithm can be approximated by the following formulae:

$$\begin{cases} Comp(p,q) = Comp(q,p) \\ Comp(p,q) = T(q,p-1) \\ T(p,0) = 1 \\ \text{for } q > 0,\, T(p,q) = \sum_{i=1}^{p} \begin{pmatrix} p \\ i \end{pmatrix} \times T(q,p-i) \end{cases}$$

Using these formulae, we get values for small-size problems which are summarized in table 5.

p & q	1	2	3	4	5	6
1	1	1	1	1	1	1
2	1	3	7	15	31	63
3	1	7	31	115	391	1267
4	1	15	115	675	3451	16275
5	1	31	391	3451	25231	164731
6	1	63	1267	16275	164731	1441923

Table 5: Approximation (upper-bound) of the complexity of the algorithm that generates all the acceptable spanning trees.

References

[1] R. Agarwal, F. Gustavson, and M. Zubair. A high performance matrix multiplication algorithm on a distributed-memory parallel computer, using overlapped communication. *IBM J. Research and Development*, 38(6):673–681, 1994.

[2] Stergios Anastasiadis and Kenneth C. Sevcik. Parallel application scheduling on networsk of workstations. *Journal of Parallel and Distributed Computing*, 43:109–124, 1997.

[3] D. Arapov, A. Kalinov, A. Lastovetsky, and I. Ledovskih. Experiments with mpc: efficient solving regular problems on heterogeneous networks of computers via irregularization. In *Irregular'98*, LNCS 1457. Springer Verlag, 1998.

[4] F. Berman. High-performance schedulers. In I. Foster and C. Kesselman, editors, *The Grid: Blueprint for a New Computing Infrastructure*, pages 279–309. Morgan-Kaufmann, 1998.

[5] L. Blackford, J. Choi, A. Cleary, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. Scalapack: A portable linear algebra library for distributed-memory computers - design issues and performance. In *Supercomputing '96*. IEEE Computer Society, 1996.

[6] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide*. SIAM, 1997.

[7] Vincent Boudet, Fabrice Rastello, and Yves Robert. Algorithmic issues for (distributed) heterogeneous computing platforms. In Rajkumar Buyya and Toni Cortes, editors, *Cluster Computing Technologies, Environments, and Applications (CC-TEA'99)*. CSREA Press, 1999. Available at `www.ens-lyon.fr/LIP/`as Technical Report 99-19.

[8] Vincent Boudet, Fabrice Rastello, and Yves Robert. A proposal for a heterogeneous cluster ScaLAPACK (dense linear solvers). In Hamid R. Arabnia, editor, *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*. CSREA Press, 1999. Available at `www.ens-lyon.fr/LIP/`as Technical Report 99-17.

[9] J. Choi, J. Demmel, I. Dhillon, J. Dongarra, S. Ostrouchov, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. ScaLAPACK: A portable linear algebra library for distributed memory computers - design issues and performance. *Computer Physics Communications*, 97:1–15, 1996. (also LAPACK Working Note #95).

[10] J. Choi, J. Dongarra, S. Ostrouchov, A. Petitet, D. Walker, and R. C. Whaley. The design and implementation of the ScaLAPACK LU, QR, and Cholesky factorization routines. *Scientific Programming*, 5:173–184, 1996.

[11] E. Chu and A. George. QR factorization of a dense matrix on a hypercube multiprocessor. *SIAM Journal on Scientific and Statistical Computing*, 11:990–1028, 1990.

[12] Michal Cierniak, Mohammed J. Zaki, and Wei Li. Scheduling algorithms for heterogeneous network of workstations. *The Computer Journal*, 40(6):356–372, 1997.

[13] D. E. Culler and J. P. Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, San Francisco, CA, 1999.

[14] J. Dongarra, R. van de Geijn, and D. Walker. Scalability issues in the design of a library for dense linear algebra. *Journal of Parallel and Distributed Computing*, 22(3):523–537, 1994.

[15] J. J. Dongarra and D. W. Walker. Software libraries for linear algebra computations on high performance computers. *SIAM Review*, 37(2):151–180, 1995.

[16] G. Fox, S. Otto, and A. Hey. Matrix algorithms on a hypercube i: matrix multiplication. *Parallel Computing*, 3:17–31, 1987.

[17] J.S. Frame, G. de B. Robinson, and R.M. Thrall. The hook graphs of the symmetric group. *Canadian Journal of Mathematics*, 6:316–325, 1954.

[18] Michael R. Garey and Davis S. Johnson. *Computers and Intractability, a Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, 1991.

[19] M. Iverson and F. zgner. Dynamic, competitive scheduling of multiple dags in a distributed heterogeneous environment. In *Seventh Heterogeneous Computing Workshop*. IEEE Computer Society Press, 1998.

[20] A. Kalinov and A. Lastovetsky. Heterogeneous distribution of computations while solving linear algebra problems on networks of heterogeneous computers. In P. Sloot, M. Bubak, A. Hoekstra, and B. Hertzberger, editors, *HPCN Europe 1999*, LNCS 1593, pages 191–200. Springer Verlag, 1999.

[21] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing.* The Benjamin/Cummings Publishing Company, Inc., 1994.

[22] M. Maheswaran and H. J. Siegel. A dynamic matching and scheduling algorithm for heterogeneous computing systems. In *Seventh Heterogeneous Computing Workshop*. IEEE Computer Society Press, 1998.

[23] H.J. Siegel, H.G. Dietz, and J.K. Antonio. Software support for heterogeneous computing. *ACM Computing Surveys*, 28(1):237–239, 1996.

[24] G.C. Sih and E.A. Lee. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE Transactions on Parallel and Distributed Systems*, 4(2):175–187, 1993.

[25] M. Tan, H.J. Siegel, J.K. Antonio, and Y.A. Li. Minimizing the aplication execution time through scheduling of subtasks and communication traffic in a heterogeneous computing system. *IEEE Transactions on Parallel and Distributed Systems*, 8(4):857–1871, 1997.

[26] J.B. Weissman and X. Zhao. Scheduling parallel applications in distributed networks. *Cluster Computing*, 1(1):109–118, 1998.